Design Implications for End-User Debugging Tools: A Strategy-Based View

Valentina Grigoreanu, Margaret Burnett

Oregon State University

George Robertson

Microsoft Research

RUNNING HEAD: STRATEGY-BASED END-USER DEBUGGING TOOLS

Corresponding Author’s Contact Information:
Valentina Grigoreanu
i.valentina.g@gmail.com
School of Electrical Engineering and Computer Science
Oregon State University
Corvallis, OR 97331

Brief Authors’ Biographies:

Valentina Grigoreanu is a computer scientist with an interest in human-computer interaction, end-user software engineering, and debugging strategies; she is a PhD candidate in Computer Science at Oregon State University and a User Experience Researcher in the Management and Administration Experience team at Microsoft.

Margaret Burnett is a computer scientist with an interest in human-computer interaction, end-user programming, and end-user software engineering; she is a Professor of Computer Science in the School of Electrical Engineering and Computer Science at Oregon State University.

George Robertson is a computer scientist with an interest in information visualization, 3D user interfaces, and human-computer interaction; he is a Principal Researcher in the Visualization and Interaction Research group of Microsoft Research.
ABSTRACT

End-user programmers’ code (e.g., accountants’ spreadsheet formulas) is fraught with errors. To help mitigate this problem, end-user software engineering research is becoming established. However, most of this work has focused on feature usage, rather than debugging strategies. If a debugging tool were to support end-user programmers’ specific debugging strategy needs, what should it take into account and how? To consider the design of such tools, this work contributes a comprehensive overview of end-user debugging strategies at four strategy levels. An example empirical study in Microsoft Excel demonstrates that this view of debugging provides useful insights, and we argue that many of these insights generalize to other environments. Our results include end-user debugging tactics and the effective and ineffective moves employed to achieve them, ten end-user debugging strategems applied to a new environment, and how these strategems were used within three contexts: by strategy used, by sensemaking step, and by debugging phase. These findings coalesce into a comprehensive overview of end-user debugging strategies and detailed implications for the design of strategy-based end-user debugging tools.
CONTENTS

1. Introduction and Related Work ................................................................. 4
   1.1. Problem Addressed and Terminology .................................................... 4
   1.2. Background and Related Work ............................................................. 5
   1.3. Research Contributions and Overview of this Paper ............................... 7
2. Study Methodology ................................................................................. 7
   2.1. Participants ......................................................................................... 8
   2.2. Procedure, Task, and Materials ........................................................... 8
   2.3. Spreadsheet Environment and Tutorial ................................................ 8
   2.4. Analysis Methodology ........................................................................ 9
3. Low-Level Strategy Items: Moves and Tactics ........................................... 9
   3.1. Some Well-Supported Tactics ............................................................... 10
   3.2. Improving Support for Tactics .............................................................. 10
4. High-Level Strategy Items: Strategems and Strategies ............................... 14
   4.1. Generalizing Ten End-User Debugging Strategems across Three Environments . 14
   4.2. The Missing To-Do Listing Strategem and Two End-User Debugging Strategies 15
5. Strategems in Context ............................................................................. 17
   5.1. Strategems in the Context of Strategies by Debugging Phase ................... 17
   5.2. Strategems in the Context of Sensemaking ............................................ 20
       Three Outlier Strategems in the Context of Sensemaking .......................... 20
       Strategems during Foraging versus Sensemaking Steps ............................ 22
       Strategems used Successfully during Schema and other Sensemaking Steps .... 22
6. Conclusions, Discussion, and Future Work ............................................. 24
Notes ........................................................................................................... 27
References ................................................................................................. 28
Figure and Table Captions ........................................................................ 34
Figures ....................................................................................................... 35
1. INTRODUCTION AND RELATED WORK

1.1. Problem Addressed and Terminology

In the United States today, there are tens of millions more end-user programmers than there are professional developers [Scaffidi et al. 2005]. End-user programmers were described by Nardi as people who (as opposed to professional developers) do not code as an end in itself, but as a means to more quickly accomplish their own tasks or hobbies [Nardi 1993]. In fact, end-user programmers often do not have professional computer science training; examples include accountants using spreadsheet formulas to keep track of a company’s budget in Excel and designers building interactive web applications in Flash.

End-user programmers’ code is widely known to be rife with errors [Panko 1998; Butler 2000; Boehm and Basili 2001; EuSprIG 2009; Panko and Orday 2005], costing companies and governments millions of dollars. For example, a Nevada city budget spreadsheet posted to the city’s website and distributed to city council members falsely showed a $5 million dollar deficit in the water and sewer fund [Nevada Daily Mail 2006]. This discrepancy delayed voting on the city’s yearly budget. Upon closer examination, the finance director found several bugs in the budget spreadsheet. This is just one of many such news stories [EuSprIG 2009].

Much of the work on creating tools to help end-user programmers eliminate such bugs has been about feature usage and mapping techniques from professional software engineering to this population. However, one area which we have recently begun conducting research into is end-user programmers’ specific debugging strategy needs. While in our earlier work we used the term “strategy” to refer to any “reasoned plan of action for achieving a goal,” in this paper we differentiate between nuances of strategy items at different levels. To do so, we employ Bates’ terminology for four strategy levels for online searching [Bates 1990].

Bates argues that search systems should be designed to make a good strategy easy to employ, by taking into account search behaviors which promote the strategic goals of searching for information [Bates 1990]. This advice also holds in designing debugging tools for end-user programming environments. Bates’ four levels of strategy are moves, tactics, strategems, and strategies. A move is at the lowest level of abstraction: “an identifiable thought or action” [Bates 1990]. A tactic is “one or a handful of moves made.” A strategem is “a larger, more complex, set of thoughts and/or actions.” Finally, a strategy is “a plan which may contain moves, tactics, and/or strategems for the entire search,” or in our case, the entire debugging session. To refer to all four of these levels, we employ the phrase strategy levels, and to refer to an item at any of the four levels, we employ the term strategy item.
1.2. Background and Related Work

In this section, we briefly summarize related work to situate our current paper within the existing literature. However, we defer detailed discussion of related work until the results sections.

Three areas of research are particularly relevant to our current focus: work on end-user programmers in general, on professional programmers’ debugging strategies, and on gender differences in problem-solving strategies. Research on end-user programmers is becoming established, ranging from automatic error detection in spreadsheets to a testing methodology for end users and to the design of web applications (e.g., [Abraham and Erwig 2007; Ayalew and Mittermeir 2003; Beckwith et al. 2006; Burnett et al. 2003; Burnett et al. 2004; Ko et al. 2004; Myers et al. 2006; Nardi 1993; Rode and Rosson 2003; Rosson et al. 2007]). Our work also has strong ties to research on novice and expert professional developers’ debugging practices (e.g., [Jeffries 1982; Kelleher and Pausch 2005; Nanja and Cook 1987; Pane and Myers 1996]), and especially on professional developers’ debugging strategies (e.g., [Katz and Anderson 1988; Romero et al. 2007]). Finally, gender differences have been observed in the use of problem-solving strategies ranging from mathematics to financial decision making, and neuroeconomics, among others (e.g., [Bandura 1986; Byrnes et al. 1999; Carr and Jessup 1997; Gallagher and De Lisi 1994; Gunzelmann and Anderson 2008; Jay 2005; Kuchian et al. 2005; Lawton and Kallai 2002; Meyers-Levy 1989; Powell and Ansic 1997]). And they have even been observed in end-user programmers’ feature usage (e.g., [Beckwith et al. 2005; Busch 1995; Ioannidou et al. 2008; O’Donnell and Johnson 2001]). Thus, in the current study, we made sure to involve an equal number of male and female participants, and to pull from the literature on end-user programming and on debugging strategies in discussing the generalizability of our findings.

Our own previous work on end-user debugging strategies has provided the background for the current study. In the upcoming few paragraphs, we use Bates’ terminology to describe that work (recall that each of those papers employs the more general term “strategy”).

For our first formative work in the direction of strategies, we mined existing end-user debugging moves data, using the sequential pattern matching data mining method [Grigoreanu et al. 2006]. This first study revealed frequencies of types of tactics employed by users in the Forms/3 research spreadsheet environment. The frequencies of tactic usage were almost identical for the unsuccessful females and the successful males. However, for this first study, we had no in-the-head data from the users. This analysis was later extended in [Fern et al. 2009] to automatically approximate some end-user debugging strategems based on sequences of moves alone, without verbal data from the users.

Our first study to take participants’ verbal assessments of their strategy items into account revealed eight end-user debugging strategems, and gender differences in either the preference or effectiveness of seven of them [Subrahmaniyan et al. 2008].
Furthermore, females’ strategems were least well-supported in end-user debugging environments.

The three studies mentioned above used an academic spreadsheet prototype (Forms/3) and academic study participants. This approach has clear advantages: the features presented in those studies can be carefully controlled, participants are fairly easy to come by, and there are few limitations on how the software can be modified for future studies. However, a tradeoff of such studies is that they have low external validity; the results pertain only to that very specific academic setting. We therefore applied this set of strategems to qualitatively code IT professionals strategems during script debugging in the commercial environment Windows PowerShell [Grigoreanu et al. 2009a]. Use of a wildcard code revealed two new strategems, and several of the strategems’ definitions were modified to generalize across end-user debugging environments. This work resulted in a set of ten strategems (listed in Table 3), which we employed as our code set in analyzing the data from the study reported in this paper.

*Testing* was one of the male effective strategems, which was especially well-supported by the Forms/3 environment. Would it be enough to encourage females to use testing? We added two tools to Forms/3 to answer this question: one for reducing the risk of employing the new testing features and another to explain how to employ the testing strategem [Grigoreanu et al. 2008]. While this approach increased females’ use of the testing strategem and had a positive effect on their confidence, it did not lead to an increase in either bugs found or fixed. Encouraging users to employ one particular effective strategem might not be the right approach. Instead, we believe end-user debugging tools should encourage the effective usage of all ten strategems.

Moves and tactics can be observed, and strategems can be extracted from the user with open-ended questions and think-aloud methods, as in the previous studies. However, the overall strategy the user is employing (if it even exists) is often unconscious, may change as progress is made toward the ultimate goal, is hard to observe, and is even harder to measure scientifically. To measure strategies, we turned to research on sensemaking, which provided us with a model of the entire problem-solving process.

Sensemaking is a term used to describe how people make sense of the information around them, and how to represent and encode that knowledge, so as to answer task-specific questions [Furnas and Russell 2005; Russell et al. 1993]. Three seminal sensemaking models include Dervin’s sense-making triangle [Dervin et al. 2003], Russell et al.’s learning loop in measuring the cost structure of sensemaking [Russell et al. 1993], and Pirolli and Card’s Sensemaking Model for Analysts [Pirolli and Card 2005]. Since the 1970’s, many other sensemaking studies have been conducted in several domains (e.g., [Sharger and Klahr 1986; Furnas and Russell 2005; Leedom 2001; Klein et al. 2006]). Stefik et al. provide a comprehensive overview of the existing sensemaking models [Stefik et al. 2002]. In one of our earlier papers, borne out of the empirical study which this paper is also based on, we derived a Sensemaking Model for End-User Programmers [Grigoreanu et al. 2009b] based on the Sensemaking Model for Intelligence Analysts [Pirolli and Card 2005]. Using this model, we observed two end-user debugging strategies: comprehensive and selective debugging [Grigoreanu et al. 2009b].
1.3. Research Contributions and Overview of this Paper

The contribution of this paper is the next step in this line of research: a comprehensive overview of end-user programmers’ debugging strategies. We analyzed the data collected for [Grigoreanu et al. 2009b] at the other three levels of strategy: moves, tactics, and strategems. We believe that this approach was fruitful in providing implications for both the improvement of existing features and the design of novel tools to help end-user programmers create more correct code. In particular, our research goals were to:

1. Examine a set of strategy data at four different levels of abstraction (moves, tactics, strategems, and strategies),

2. Investigate the use of strategy items in the context of the purpose for which they are employed: sensemaking step and debugging phase (bug finding, bug fixing, and evaluating a fix).

This paper presents three main contributions. (1) In addressing the two goals above, it is the first to provide a comprehensive overview of strategy item usage during end-user debugging. We analyzed our participants’ data at four strategy levels and in the context of two purposes. (2) Analysis at each level and for each purpose resulted in a different type of implication for the improvement of one of the most popular end-user programming environments: Excel. These implications ranged from iterative improvements to existing tools to the design of novel tools for Excel. The range of implications highlights the importance of investigating multiple strategy levels and contexts in the design of tools. (3) We discuss the generalizability of these findings to other programming environments.

We begin in Section 2 by presenting the study methodology and analytical methods. Next, we report our results on the comprehensive view of end-user debugging strategy items at four levels and in two contexts in Sections 3-5. This is also where we integrate the resulting implications for design, reveal opportunities for future research, and discuss the generalizability of our findings. Finally, we reiterate our conclusions in Section 6.

2. STUDY METHODOLOGY

The data we analyzed for this study came from an earlier experiment which examined how traces of Sensemaking model traversals can be used to better understand end-user debugging strategies [Grigoreanu et al. 2009b]. In this second round of analysis, we complete those findings by also examining the other three strategy levels (moves, tactics, and strategems), in the context of both individual sensemaking steps and also by debugging phase. While we have reiterated the main parts of our study methodology here, the reader should refer to [Grigoreanu et al. 2009b] for a more complete overview.
2.1. Participants

Our eight participants (four men and four women) were undergraduate and graduate students at Oregon State University who had experience with Excel spreadsheet formulas. They received a $20 gratuity for their participation. None of the participants were computer science or electrical engineering majors, and they did not have any programming experience beyond the requirements of their majors (for example, a Business class on Business Application Development). In this study, we examined the four most and least successful participants’ strategy items in much detail. We refer to these as Participant SM (successful male), SF (successful female), UM (unsuccessful male), and UF (unsuccessful female).

2.2. Procedure, Task, and Materials

We used a think-aloud methodology [Ericsson and Simon 1984] to capture end-user programmers’ in-the-head strategic activities data. After signing the required paperwork, each participant answered interview questions about their spreadsheet background, got a brief tutorial about the Excel auditing features, and practiced thinking aloud on a short warm-up task.

The participants’ actual task is shown in Figure 1; the figure shows the paper handout we provided them to describe how the spreadsheet was supposed to work. This spreadsheet was obtained from the EUSES Spreadsheet Corpus of real-world spreadsheets [Fisher and Rothermel 2005]. It was complex, containing two worksheets and 288 formula cells. Each participant had 45 minutes to “make sure the spreadsheet is correct and, if [you] find any errors, fix them.” We recorded participants’ voice, facial expressions, and computer screen as they debugged, and also saved their final spreadsheet.

FIGURE 1 ABOUT HERE

We seeded the spreadsheet with a total of ten varied real bugs harvested from the spreadsheets of [Beckwith et al. 2007]’s participants (Seattle-area adult spreadsheet users). These bugs were therefore realistic in terms of the kinds of bugs real end users create: six were inconsistent with other formulas in the same row or column (e.g., a missing student in a class total), three had been propagated by their original authors over the entire row or column (e.g., using the “>“ operator instead of “>=“), and one was not part of any group of similar formulas but was nevertheless wrong (it counted lab attendance as a part of the total points when it should not have). The number of bugs found and fixed by each of the most and least successful participants are listed in Table 1.

TABLE 1 ABOUT HERE

2.3. Spreadsheet Environment and Tutorial

The environment for the study was Microsoft Excel 2003. To make sure the participants were familiar with some of Excel’s debugging features, we gave them a
hands-on pre-session tutorial. The tutorial was only a tour of Excel’s auditing features (arrows, tooltips, evaluate formula, error checking, and the watch window), and did not include any strategy hints. The participants were given time to practice using these features during the mock practice task and were allowed to use any Excel features they wanted during the actual task.

2.4. Analysis Methodology

Transcripts of the participants’ think-aloud verbalizations were coded qualitatively in three passes: debugging phase, sensemaking step, and strategem employed. One researcher first labeled the phase changes of all eight participants (“bug found”, “bug fixed”, and “evaluating fix”), since these were objective. The most and least successful female and male participants were chosen based on these codes. Two researchers then coded these four participants’ transcripts with the additional two sets of codes.

For the strategy codes, we used the ten generalized End-User Debugging Strategy codes from [Grigoreanu et al. 2009a], which are listed in Table 3. For the End-User Debugging Sensemaking codes, we used the code set derived in [Grigoreanu et al. 2009b]. In that prior work, we employed eight of the nine sensemaking steps as codes plus two main loops: external data sources (this is the one we did not code for, since it referred to all of the information available to the users), shoebox (data that a participant deemed relevant enough to “touch” in the spreadsheet or study environment), evidence file (extracted from the shoebox, data that attracted a participant’s interest enough for follow-up), schema (a structure or pattern a participant noticed as to how cells or information related), hypothesis (a tentative idea about how to fix a particular bug based on the participant’s schema), presentation (the work product), reevaluate (making sure that a formula change was in fact correct after changing a formula), the tangential Environment Loop (a sensemaking loop for figuring out the environment), and the tangential Common Sense and Domain Loop (a sensemaking loop for answering questions about common-sense and domain questions). For more details about how we arrived at these codes, please refer to [Grigoreanu et al. 2009b].

For the current study, we coded as follows: while referring back to both the participants’ verbalizations and their screen captures for context, the coders first independently coded 20 minutes of the most successful female’s transcript, and achieving 93% inter-rater reliability (calculated using the Jaccard index) on the Debugging Strategems codes. Recall also that we had achieved an 84% agreement on the Sensemaking codes [Grigoreanu et al. 2009b]. The two researchers then split up the remaining videos and coded them independently, since the level of agreement reached showed that the set of codes was complete and unambiguous for this data set.

3. LOW-LEVEL STRATEGY ITEMS: MOVES AND TACTICS

We first consider moves and tactics together in the design of end-user debugging tools. Recall that a move is the smallest unit of a strategic activity. In this work, we
consider it simply using an environment feature. A tactic is the use of one or more moves with the purpose of more quickly and accurately finding or fixing a bug. Most commercial environments contain hundreds of features, so we will not list them here. Instead, we present some selected tactics employed by the participants and the moves used to implement them.

3.1. Some Well-Supported Tactics

Many tactics were well-supported in Excel; the participants’ moves to implement those tactics were effective. In this section, when we use the term “effective” to describe moves, we mean that they were performed using features designed specifically for that tactic, rather than finding click-intensive workarounds (which we call “ineffective” because they cost users an unnecessarily high number of actions). Whenever we mention ineffective moves, we contrast them with effective ways of performing the same tactic.

Here are two examples of tactics which allowed us to identify moves that participants were able to use to good effect: “viewing dependencies” and “judging the correctness of an error checking warning.”

Spreadsheets employ a dataflow-oriented execution model: each used cell has an output value and other cells can use that output value to produce further output values. It is therefore not surprising that Excel has several features to help users view cell dependencies. These include “Trace Precedents”, “Trace Dependents”, and the color-coded outlines of formula precedents attained by double-clicking a formula cell. All four participants used these features to effectively find cells’ precedents and dependents. For example, Participant SM traced the precedents of a cell which contained an error. The error had been propagated from a precedent, which the red arrows (usually blue) he brought up took him to. All four participants also often relied on the precedent arrows to help them parse complicated formulas.

Judging the correctness of an error checking warning is an important tactic to employ when using Excel’s Error Checking functionality to find and fix bugs. Participant SF demonstrated how to do this in three distinct ways. First, she tried to understand why a cell was getting an “inconsistent formula” warning by scrolling up and down through the formulas that were supposed to be consistent using the arrow keys. The formulas looked consistent. Second, she traced the precedents for the cell with the warning and the precedents for cells it should have been consistent with; the dependencies were also consistent. By this point, she had decided that she should not follow the warning, and that the formula was indeed consistent, but was puzzled by the reason behind the warning. In response, the system was transparent enough to allow her to derive yet another correct answer. Participant SF noticed the “Copy Formula from Left” action button, which made her realize why the Excel inconsistency warning was flawed in this case where a row of consistent formulas intersected with a column of consistent formulas: “It says ‘Copy from Left’ instead of ‘Copy from Above’, so those formulas aren’t inconsistent at all.”

3.2. Improving Support for Tactics
While many moves were effectively employed to implement a tactic, sometimes, the tactics were good but the moves were not. Table 2 reveals several successful tactics employed by our participants, but which were implemented using an ineffective combination of moves. Software users have been reported in the past to often employ ineffective moves in performing computer tasks [Bhavnani et al. 2001].

TABLE 2 ABOUT HERE

The list of tactics and moves presented in this section is certainly not exhaustive, but it does provide useful low-level implications for design. How could Tactic 1, finding formula cells, be better supported in Excel? (See Table 2 for the ineffective moves employed by our participants.) There are better ways of finding formula cells in Excel. For example, the sequence of moves “Find & Select \(\rightarrow\) Formulas” selects all formula cells in the worksheet, while “Find & Select \(\rightarrow\) Constants” selects the remaining used cells. A different set of moves the participants could have performed would have been to toggle the “Show Formulas” option on, which would have switched the spreadsheet view to one which shows formulas in the cells as opposed to their calculated results. Both of these approaches have the same disadvantages: the features are hidden, and simply selecting the formula cells does not make them jump out.

**Implication for Design 1:** Finding formula cells (or other code to be debugged) is an important tactic in the debugging process, which all four participants employed. Tools should provide an easily accessible button to highlight all applicable code in an obvious way.

Researchers have started creating some tools to highlight formulas by surrounding formula cells with a special border. For example, some tools overlay a border around areas with consistent formulas to help visualize broken areas (e.g. [Sajianemi 2000]) and others color the border of formula cells for the purpose of reflecting the testing coverage achieved for that cell (e.g., [Burnett et al. 2003]). Our evaluation of moves and tactics in Excel supports such efforts, and even provides an advantage of broken areas with borders over triangle warnings: despite both being valid ways of visualizing inconsistencies, the former also makes obvious where the formulas are.

This takes us to Tactic 2, finding inconsistent formulas. Investigating the moves employed to accomplish this tactic by all four participants revealed some shortcomings of the error checking tool. The tactic of looking for inconsistent formulas would have been fruitful: six of the ten bugs were inconsistency bugs, and five of those were revealed by Excel. Unfortunately, our participants employed ineffective moves for accomplishing this tactic.

As Table 2 shows, one set of moves used by all four participants was to hit the arrow keys while watching the formula bar, looking for changes in the formula. One of several much more efficient set of moves to perform in Excel would have been to, while running Error Checking in the foreground, click the “Options…” button and uncheck everything except “formulas inconsistent with other formulas in the region.” Three of our four participants did not look for these more effective options and the fourth (Participant SF)
only did so after clicking through almost all of the 202 error checking warnings provided by Excel by default on this spreadsheet. When she finally discovered how to ignore all but the “inconsistent formula” warnings, Participant SF contently exclaimed, “Oooh. You can pick the errors if you want to. And it only shows you the ones that are [inconsistent]… Oooooh!” This high false-positive rate might also have been the reason for how little participants employed the error checking tool. Of the 202 warnings, 197 (or 98% of them) were false-positives! Of the eight inconsistent formula warnings, however, more than half (five) were truly bugs in the formulas; a much better false-positive rate.

**Implication for Design 2:** Too much feedback from the system about what might be (or lead to) an error may be as unaccommodating as having none. While automatic error checking algorithms are improving and detecting more and more types of suspicious code, default settings should rank the likelihood of each type of warning to reveal a bug, and only show the most likely warnings by default. Formatting and other warnings which are less likely to cause a bug should be made accessible by the user optionally.

As described earlier, viewing cell dependencies was a well-supported tactic in Excel. However, as with the error checking tool, a closer examination of how dependency arrows were used for the purpose of implementing Tactic 3 also revealed possible improvements to the tool. As Table 2 describes, trouble arose when dependencies were between worksheets. Showing inter-worksheet relationships is also a shortcoming of many research systems. In Excel, when a user brings up arrows for a cell which has precedents/dependents in a different worksheet, the cell points to a small table icon (see Figure 2). Three of the four participants encountered this icon, but none understood what it meant. Participant SF first hovered over it and then simply ignored the icon, stating “There’s a little box, but I don’t know what that means.” Participant UF took the arrows down immediately. Participant UM, also first hovered over the table, perhaps expecting a tooltip. Next, he double-clicked the icon which only selected the cell behind it, followed by exclaiming, “Oh, oh!”

**FIGURE 2 ABOUT HERE**

Furthermore, only half of the participants moved on to the second worksheet, late in the task (Participants SF and UM both first started examining formulas in the second worksheet at minute 32, and neither found the bug in it). Thus, while arrows were very popular with all four of these participants, this detailed examination of the moves used for a particular tactic led to several observations for how viewing cell dependencies could be improved.

**Implication for Design 3:** While intra-worksheet dataflow relationships are typically well-supported in spreadsheet environments, visualizations of inter-worksheet are not. In addition to being understandable, these tools should also allow users to easily navigate to those other worksheets.

Several studies have found that end-user programmers reuse working code created by themselves and others (e.g., [Dorn et al. 2007; Bogart et al. 2008]). Three of our four
participants attempted to employ code reuse to help them fix errors. Three examples of this include copying a consistent formula over an inconsistent one (Participants SF and SM), searching Excel Help for useful formulas to use in fixing a bug (Participants SM and UF), and looking for other formulas in the open spreadsheets which can be applied in the current situation (Participant SM). Just as Participant SM did here, inspecting code in other scripts also helped a successful female participant correctly fix a bug she had previously fixed incorrectly in the Windows PowerShell scripting environment [Grigoreanu et al. 2009a], and this pattern has also been observed with professional developers [Ye and Fischer 2002].

**Implication for Design 4:** Code reuse for the purpose of fixing bugs needs to be improved. In particular, searching for related code that is not in spatial proximity (as is the case with an inconsistent formula) is especially hard. One set of moves which has been used successfully to fix a bug is to recognize related code while skimming other files. End-user programming environments should facilitate this process by listing related tokens (such as, the types of formulas used in this spreadsheet, or the formulas used in any spreadsheet in a particular directory).

Finally, the two female participants especially turned to the environment to figure out what they can do next (Tactic 5) to debug the spreadsheet, after they thought they had done everything they could. Examples of this tactic involved clicking features in the Audit Toolbar and searching for Help. Unfortunately, this exploration time always came very late in the debugging process, when they were no longer very useful. For example, the first time Participant SF read a comment in one of the column titles with a description of what that formula was supposed to do was at minute 34! This was only after she believed she had run out of things to do, even though it would have been useful for her to do so while judging the accuracy of that formula (which actually contained an error she never found).

This tactic is highly related to search tools of the kind recently found to be needed by professional developers for finding appropriate knowledge, resources, and skills [Aranda and Venolia 2009; Venolia 2006]. This need to view related debugging information in the current context was also important to IT Professionals using PowerShell: for example, while inspecting code, the participants wanted to easily access incremental testing information in context (such as by hovering over a variable after running the code to a breakpoint) and, while examining error messages which resulted from running the script, participants wanted a shortcut to code related to those messages [Grigoreanu et al. 2009a]. Another approach would be for the environment to generate a list of to-do items and to provide all of the relevant information by each item, solving both the problem of deciding what to do next and seeing what information can be checked next within that context.

**Implications for Design 5-6:** The environment should recommend items to check. It should also provide relevant subtasks (or information which could help the user debug a particular part of the code) to the user on-the-fly in that context. Currently, the information is scattered about in menus and submenus, on paper,
in emails, and in comments, among other places, and important details and resources relevant debugging are easy to overlook.

This is only a small sample of the tactics employed by end-user programmers during debugging. Our formative work on end-user debugging strategies employed sequential pattern mining algorithms to look for common short (one to four) sequences of events performed by users, and we found 107 such frequent patterns [Grigoreanu et al. 2006]. Many of these common sequential patterns of events might reveal new debugging tactics. A more comprehensive set of end-user debugging tactics, like Bates’ search tactics [Bates 1990], has yet to be compiled.

The findings presented in this subsection have implications for usability research. Even this small sample of seven tactics (two well-supported and five poorly-supported) helped us uncover several Excel usability strong points and problems, leading to detailed implications for the design of debugging tools. Furthermore, the resulting implications are likely to generalize to most other debugging environments, since most include at least one of the types of features addressed here (dataflow dependencies, error checking capabilities, areas of related code, multiple worksheets or files, reusable code, comments and other cell-specific information, etc.). Thus, examining the effective tactic a set of features was used toward proved to be a promising way to quickly improve even one of the oldest and most popular end-user programming environments.

4. HIGH-LEVEL STRATEGY ITEMS: STRATEGEMS AND STRATEGIES

A strategem is a complex set of thoughts and/or actions, while a strategy is a plan for the entire task which may contain strategems [Bates 1990]. In this subsection, we explore the debugging strategems used by end-user programmers, the purposes for which the strategems are used, and two strategies identified based on the participants’ ways of traversing the sensemaking model.

4.1. Generalizing Ten End-User Debugging Strategems across Three Environments

Table 3 presents a set of ten strategems [Grigoreanu et al. 2009a] and their ties to gender and success. These strategems previously generalized across two very different end-user programming environments and populations: students using the spreadsheet prototype Forms/3 and IT professionals using the commercial scripting environment Windows PowerShell [Grigoreanu et al. 2009a]. Could we apply this same code set to a third environment?

TABLE 3 ABOUT HERE

As the very high inter-rater reliability (93%) presented in Section 2.4 showed, the code set from the PowerShell study was both complete and unambiguous for this new
data set. Despite having a wildcard “Other” code during the content analysis phase, we never had to use it: the code set was saturated. While the list of strategems is likely to still be incomplete, this generalization of the codes across three environments (Microsoft Excel, Windows PowerShell, and Forms/3) and populations lends credence to their good coverage and unambiguous definitions.

Figure 3 shows the total minutes spent by the four participants using each strategem. Notice that while the code set was saturated, two of the strategems were absent: control flow and to-do listing. It is not surprising that control flow was absent as a strategem, because there is no explicit control flow in spreadsheets; rather, execution sequence is derived by the system from dataflow dependencies. However, it is interesting to consider control flow one level up from the standpoint of what it is often used for, namely automating repetition. Participant SM wanted an effective tactic for repetition; his view was that he wanted to employ a “for loop” to fix one of the bugs, had it been available. Furthermore, expert users sometimes resort to additionally applying languages such as VBA [VBA 2009] and its simplified version for Excel spBasic [spBasic 2007] as a control flow engine for spreadsheets. However, users who do not use such additional tools often have to resort to writing tediously lengthy formulas. For example, since Participant SM found no way of doing a for loop in Excel, he wrote a lengthy formula which considered each cell of a row individually. Peyton Jones et al.’s proposed user-defined functions that operate on matrices are one possible approach to introduce repetition automation in purely declarative environments [Peyton Jones et al. 2003].

**FIGURE 3 ABOUT HERE**

*Implication for Design 7:* Even for declarative end-user programming environments, a feature for effectively implementing repetition is needed to reduce the tedium of creating repetitive code, without having to learn a new language and environment.

The absence of the to-do listing strategem in this study is harder to explain. This strategem was used by participants in conjunction with the other strategems to keep track of parts of the code that are “done” vs. “to-do” vs. “unchecked.” To-do listing was reported in both the Forms/3 spreadsheet environment [Subrahmanian et al. 2008] and in the PowerShell environment [Grigoreanu et al. 2009a], and its use has also been investigated in professional developers’ work [Storey et al. 2008].

4.2. The Missing To-Do Listing Strategem and Two End-User Debugging Strategies

A possible reason for the lack of to-do listing in Excel might be the environment’s lack of moves that would make the strategem convenient. While neither the Forms/3 nor the Excel environment provides direct support for to-do listing, both contain features which can be repurposed for it. In Forms/3, participants repurposed a prevalent testing feature (a checkbox on each cell) to be a judgment about the formula’s correctness, instead of the value’s correctness as intended. In Excel, participants could have changed the background color of a cell or left a comment, for example, yet no participant did so.
There are several differences between the two spreadsheet tasks and the two environments which could account for this. First, the Excel spreadsheet was highly formatted, containing one blue column, one yellow column, four gray columns, 30 rows with alternating colors, three different font colors, 46 cells with bold fonts, five underlined fonts, many different font faces, and all borders delimiting spreadsheet regions. Thus, had formatting information been used to keep track of the to-do list (as is now even more encouraged with the new “Cell Styles” feature in Excel 2007), there would have been a big loss of formatting information in this spreadsheet. The Forms/3 spreadsheet did not contain any formatting. Thus, when the checkbox feature was used in Forms/3, there was no loss of formatting information when the border colors and/or background colors also changed. A second reason for this difference in participants’ propensity to repurpose features for to-do listing might have been the size of the spreadsheet given. The Forms/3 spreadsheet only contained 13 formula cells to check off. In this study, our spreadsheet contained 288 formula cells, which makes the structure of the spreadsheet harder to parse in figuring out a list of items to-do.

**Implication for Design 8:** Repurposing other features to support to-do listing is work-intensive for today’s complex end-user programs. Instead, a tool is needed to directly support this strategem by taking into account consistent areas of the code, while providing the flexibility of adding and removing to-do items, all without code formatting information loss.

Just as it helped to examine moves within the context of tactics, we believe that, for the design of tools, it is also useful to observe strategems within the context of strategies. How might the missing to-do listing strategem have been useful if employed as a part of the two successful end-user debugging strategies?

Our earlier sensemaking work revealed two major strategies employed by end-user programmers during spreadsheet debugging: comprehensive and selective debugging [Grigoreanu et al. 2009a] (see Table 4). These strategies and their gender ties were consistent with results on the comprehensive and selective information processing styles reported by the Selectivity Hypothesis [Meyers-Levy 1987]. While the comprehensive strategy held many advantages, such as finding and fixing the easier bugs early on, it also held disadvantages. Three of those disadvantages included: (1) sometimes overlooking less prominent formula cells by mistake, (2) forgetting about bugs found earlier in the task but not followed up on right away, and (3) sticking with the strategy even after it ceased to be successful for finding and fixing bugs. Support for to-do listing may help address all three of these downsides of the comprehensive strategy.

**Implications for Design 9-10:** A to-do listing tool should be lightweight, automatically generating a list of to-do items directly from the code itself, to make sure none of the code is overlooked during comprehensive processing.

Being able to mark complicated code for later follow-up might help comprehensive strategy users return to those bugs later on. This might also help
comprehensive strategy users switch to other strategies when the latter has ceased to be successful.

A to-do listing tool might also help participants who prefer the selective strategy. For example, Participant SM, whose selective approach enabled him to fix the hardest bug, missed out on a lot of information that enabled Participant SF to spot and fix several bugs early using a comprehensive strategy. If end-user debugging tools could support lightweight switching among both strategies, doing so may encourage end-user programmers to make the switch when their approach becomes less productive.

Implication for Design 11: To-do listing support might also help selective strategy users switch to other strategies when their approach has ceased to be successful for finding and fixing bugs. Furthermore, it will allow them to keep track of their selective progress. Thus, such a tool should be compatible with both the comprehensive and selective strategies.

Just as with tactics and moves, our report of the two strategies might not be exhaustive. Researchers studying professional programmers’ debugging have uncovered several strategies, some of which might also apply to end-user programmers. For example, Katz and Anderson classified strategies into two types: forward reasoning and backward reasoning [Katz and Anderson 1988]. Forward reasoning strategies are those where the participants’ reflection starts from the code itself: comprehension (similar to our comprehensive strategy) and hand simulation (evaluating the code as if they were the computer). Backward reasoning strategies involved starting with a noticeable incorrect behavior of the code, and working backward to the part of the code which might have been the cause for the fault: simple mapping (proceeding from error message directly to the related code) and causal reasoning (searching backward from the program’s output). We present some more strategies observed in this study in Section 5.1.

5. STRATEGEMS IN CONTEXT

Strategems are a particularly promising strategy item to examine for the purpose of tool design: unlike with low-level moves and tactics, they are much more directly generalizable, and unlike strategies, they are low-level enough to lead to very specific implications. Thus, in this section, we further examine strategem usage in two new contexts: by debugging phase (an overview of the debugging process) and by sensemaking step (a model for the overall problem-solving process).

5.1. Strategems in the Context of Strategies by Debugging Phase

A strategem was rarely used in isolation. In fact, a strategy is defined as being a series of strategems used together. Thus, we expected examining patterns of strategems used together to reveal additional end-user debugging strategies. Also, in earlier work, we observed that PowerShell scripters’ success with a strategem appeared to depend on the debugging phase at which it was being used [Grigoreanu et al. 2009a]. Therefore, in this
paper, we differentiate between strategies used for the different debugging phases (bug finding, bug fixing, and evaluating fixes).

How did our participants employ strategems together successfully in the context of strategies at each phase? To answer this question, we narrowed our focus to only the successful sensemaking loop traversals (i.e., those ending in a correct find or fix) from the successful participants’ transcripts. We then broke those successful traversals down further by debugging phase: a bug find, a bug fix, or the evaluation of a fix.

To find bugs, Participant SF primarily employed two systematic patterns, or strategies. For the first half of her task, she relied heavily on a pattern containing four intertwined strategems: specification checking to understand what a part of the spreadsheet was supposed to do, spatial to examine areas containing similar cells, code inspection to run through the formulas in that spatial area and make sure the formulas are consistent, and dataflow to see how those cells depended on other cells in the spreadsheet. The following excerpt from her session exemplifies this strategy, which we called the consistency checking strategy:

SF: “It uses Box A [refers to an outlined region in the Specification handout] to determine the letter grade and GPA <note: specification checking>. So we can look at the GPA in column H, and we go down to see if there are any major differences between the formulas <note: spatial employment of code inspection>. And checking, their dependents, which would be the average, the highest, and the lowest scores... And precedents, which would be looking things up from the average and the GPA table <note: dataflow>.”

In addition to heavily relying on these four strategems in tandem, she also had seven instances of the other strategems: prior experience in knowing that cells pertaining to a spatial region should be checked for consistency, feedback following in looking at an error checking warning, help in reading the warning associated with a suspicious cell, and testing in making judgments about values’ correctness. Thus, she relied most heavily on four of the strategems, but used all eight strategems at some point during bug finding.

But Participant SF relied on a different pattern of strategems in tandem during the second half of the session for bug finding: a core set of four strategems plus a variable fifth strategem. For the core set, she used feedback following and spatial to systematically walk through cells only paying attention to those which expressed the formula inconsistently with others in the area, help to understand what made those cells suspicious, and code inspection and spatial to judge whether or not the green triangle feedback should be ignored. We will call this strategy simple mapping since it is identical to Katz and Anderson’s strategy for finding bugs based on the software’s error messages [Katz and Anderson 1988]. This strategy also included a variable fifth strategem from the set {specification checking, dataflow, prior experience, testing}. Thus, she ultimately used all eight strategems in her second bug finding approach as well. The following quote comes from a part where dataflow was the variable strategem in her pattern:
SF: “So, I... that one’s inconsistent. <note: feedback following and spatial> [Reads the warning] ‘The formula in this cell differs from the formula in this area of the spreadsheet. Click on help with this error.’ [The help is slow to open up] It sits there with a nice box. [Closes help box] That doesn’t work. So renew, making sure they are all just unprotected formulas. ‘Omits adjacent cells.’ Which one omits? E6. [Inspects formula which also highlights precedents for which cells it should include] It doesn’t need to include any adjacent cells. <note: code inspection and dataflow> ‘Numbers stored as text,’ ‘unprotected formulas.’ Moving all the way through, looking for something that says ‘inconsistent formula.’ <note: continues with feedback following and spatial>“

Turning to the bug fixing phase, Participant SF fixed all six bugs during the first half of the task, using the consistency checking strategy.

Finally, in evaluating her fixes, she looked at cells’ output values (testing) and/or formulas (code inspection), often comparing them to others in the area (spatial). We called this the compatibility checking strategy since it involves bringing in as much related information as possible and making sure that there are no discrepancies in their compatibility. For example, after using spatial, code inspection, and dataflow to find and fix a bug, Participant wanted to see a change in the output value, and she did:

SF: “The average changes. <note: Testing>“

Participant SM found half the bugs based on those cells’ output values not matching what the formula should give (i.e., using testing and code inspection), and half while trying to fix a different bug (through spatial, code inspection, and dataflow). His use of the latter pattern is once again identical to the female’s consistency checking strategy. And, the former pattern is once again identical to one of Katz and Anderson’s professional developers’ debugging strategies: the causal reasoning strategy [Katz and Anderson 1988]. And, and here is an example of the former strategy:

SM: “Looking here right away I see a couple what look like problems. <note: testing> [Noticed a faulty output value of “F” in G12. He pauses and stares at G12’s formula.] <note: code inspection> I’m trying to see here...I’m looking at the letter grade and average and it looks like the first one isn’t associated correctly.”

Participant SM fixed bugs using the selective strategy, sticking with the latest bug find until it was either fixed or until he found a new bug to follow up on. When the bugs he was trying to fix were inconsistency bugs, he also continued to use spatial, code inspection, and dataflow to fix them). For the other bugs, during successful bug fixing traversals, he relied most heavily on a combination of code inspection, dataflow, and help, with a bit of specification checking and spatial. Finally, he used the same strategems as Participant SF for reevaluating bug fixes. For example, here is what he said and did as he fixed the hardest bug,
SM: “In the grading it says students must attend 70% of the labs in order to pass the course. <note: specification checking> [Asks the researcher a question:] Are, uh, is it possible to be waived from the labs? <note: help> [The researcher tells him to do the best he can with what he has been given:] Ok, I will not go with that, then. [Looks at formula he was editing] [Pauses and thinks] <note: code inspection> I’m supposed to assign a number to that, each one that is waived. <note: dataflow> So I know how many to look for. I’ve got to do that. Just got to find a way to make that true. [goes back to editing formula for AG 21] Ok, it mostly works for waive now. Lets test if there is a waive or not. [Looking at values] I tested all of them so... <note: testing>“

The quote above is a reminder that the help strategem was not always about asking the software for help on features. Sometimes, help involved asking Google or other people. Also, when the participant received answers, they often led to assumptions on which participants based further actions: here, Participant SM assumed that labs could not be waived and fixed the bug with that assumption in mind. All of the end-user debugging strategies we have observed these successful participants employ are reiterated in Table 4.

Implications for Design 12-15: This subsection has several new end-user debugging strategies for finding bugs (consistency checking, simple mapping, causal reasoning), fixing bugs (consistency checking), and evaluating those fixes (compatibility checking) to be supported by tools.

End-user debugging tools should remain flexible enough to also support both of the strategies we identified earlier: comprehensive and selective debugging.

Some strategems played a central role in each strategy, as pointed out in this section, and those groups of strategems can be used as a starting point for the tool’s design.

However, the other strategems should also be supported at least in a secondary way in the context of each strategy this section revealed.

5.2. Strategems in the Context of Sensemaking

Another approach to derive strategy-based implications for design is to look at strategy items in the context of sensemaking about a spreadsheet’s correctness. This second approach allows us to look at the more immediate purpose for which to use particular strategems.

Three Outlier Strategems in the Context of Sensemaking

Recall from Figure 3 that, of the strategems used to make sense about a spreadsheet correctness, most were used for about 20 minutes total by the four participants together. However, three strategems were different: code inspection, which was used extensively, and proceeding as in prior experience and specification checking, both of which were used very lightly.
In aggregate, our participants used *code inspection* about 70 minutes in total, more than three times as much as any other strategem. Further, as Figure 4 shows, participants used *code inspection* for all six of the main loop’s sensemaking steps as well as in both of the other two loops. *Code inspection* was popular in our previous studies as well (e.g., questionnaire answers revealed that *code inspection* and *testing* were used by 33% more participants than other strategems in [Subrahmanian et al. 2008]), but this study’s time-on-strategem data shows that the extent of its popularity amounted to outright dominance.

**FIGURE 4 ABOUT HERE**

There are at least three possible explanations. One of those is simply that *code inspection* is relatively ubiquitous because it complements other strategems. For example, using *dataflow* order participants are likely to inspect formulas as they move from one cell to another. Another possibility is that the phenomenon may be particular to Excel. Excel is not set up for *code inspection*: users can see only one formula at a time unless they switch entirely out of the output value view. Perhaps the lack of support for *code inspection* in Excel simply required our participants to spend more time in it than they would have in other environments in order to make any progress. A third possible explanation is that *code inspection* may be the strategem with which the participants were most familiar, and therefore tended to rely on the most. This would be consistent with a previous study, in which females reported that they relied on the familiar feature of formula editing because they thought other features would take too long to learn [Beckwith et al. 2005]. The findings lead to three opportunities for future research, implications 1-3 in Table 5.

**TABLE 5 ABOUT HERE**

Turning to the lightly used strategems, the lack of *proceeding as in prior experience* (only about five minutes in total) may have been simply a matter of participants failing to verbalize. Having recognized how to proceed from prior experience, participants may have quickly moved ahead to execute their changes. This strategy was mostly verbalized during early sensemaking steps (*shoebox* and *evidence file*), and in making sense of the *environment*. The participants’ use of *prior experience* while making sense of the *environment* is reminiscent of the needs for “closeness of mapping” (how closely the notation corresponds to the problem world) and “consistency” (after part of the notation has been learnt, this refers to how much of the rest can be guessed) in the design of environments: two of the cognitive dimensions of notation [Greene and Petre 1996]. This led to several more opportunities for future research (see Table 5, implications 4-5).

Like *prior experience*, participants did little *specification checking*, only about ten minutes in total. This could be because, in order to do this, participants would have had to switch back and forth between the paper handout specification and the spreadsheet. Also, perhaps due to the specifications being on paper, *specification checking* was used mostly during the *shoebox* sensemaking step for debugging (see Figure 4).

**Implication for Design 16:** End-user programming environments need to support specification checking in a lightweight way, especially for the shoebox “data
Specifications are one type of relevant information which can be tied to task items as described in Section 3.2.

Strategems during Foraging versus Sensemaking Steps

It is useful to note that the list of strategems and their definitions, culminating in those presented here from [Grigoreanu et al. 2009a], covered all of the sensemaking steps in this study (see Figure 4). The early steps of the Sensemaking model, which include shoebox and evidence file are a part of a sensemaking subloop called the “foraging subloop,” while schema, hypothesis, presentation, and reevaluation happen during the “sensemaking subloop” [Pirolli and Card 2005]. The figure shows that, as participants progressed through the foraging subloop and the sensemaking subloop, the variety of strategems used at each step also decreased: shoebox (8), evidence file (8), schema (8), hypothesis (6), presentation (6), reevaluate (4).

Implications for Design 17-18: Information foraging tools for end-user debugging environments should support all debugging strategems.

Tools to help users organize their findings should also support all strategems.

The dominance of code inspection during the hypothesis and presentation steps is surprising (see Figure 4); code inspection was used more than twice as much as all the other strategems put together during those steps. It is unlikely that code inspection is the only reliable strategem for hypothesizing about a bug fix and actually making the fix. One strategem that can be used in Excel is to follow the environment’s feedback about how to fix a bug: feedback following. For example, when a formula is inconsistent with the others in that column, or if it otherwise raises a red flag, Excel adds a small green Error Checking triangle to the top-left corner of that suspicious cell. It then gives the participant the option to fix the bug by clicking on the “copy from above” button, for example, to make it consistent with the rest of the column.

Recall that six of the ten bugs were inconsistency bugs, five of which Excel found. It is therefore surprising that feedback following only accounted for about 2% of the participants’ presentation step, and less than 1% of their hypothesis step. While one possibility is that feedback following was used so little because it was efficient (it only took a button click, after all), so the total time could have been short despite its frequent usage, this was not the case. The participants had very low counts of feedback following during hypothesis and presentation. In fact, only Participant UM had any such instances at all, and his count was one for each. Thus, we believe the real reason for which feedback following was scarcely used, especially for those later sensemaking steps, was because of the high number of false-positives the error checking tool brought up by default. This leads to Implication for Future Research 6 in Table 5.

Strategems used Successfully during Schema and other Sensemaking Steps

While we have argued that all strategems should be supported for the shoebox, evidence file, and schema sensemaking steps, an understanding of which strategems the successful participants employed at each step can give tool designers an idea of which
strategems to make the focus of each tool. We first provide an example of how tool designers can use these findings to support the schema step, and then also list the most successful strategems for the other steps.

The schema step was particularly troublesome for our two unsuccessful participants. After spending from half to two-thirds of their time gathering information, these unsuccessful participants got stuck in the schema step for lengthy periods of time, trying to organize it [Grigoreanu et al. 2009b]. Successful participants, however, had faster transitions in and out of this sensemaking step. Thus, it appears that strategy support for the schema step may be especially useful in helping increase users’ debugging success.

**Implication for Design 19:** Supporting the schema step is particularly important, since unsuccessful participants tended to get stuck on creating a structure for the information they collected.

How did the successful participants successfully create a structure by organizing the information they collected? To answer this question, we narrowed in on the strategems used during a successful debugging session, as defined by leading to a correct bug find or fix. The following four strategems each accounted for at least 10% of successful schematizing: code inspection (38% of schema time), spatial (29%), testing (14%), and dataflow (10%). While these four strategems were most common during the Schema creation sensemaking step, the remaining strategems were also used slightly, and therefore should be supported in a secondary way.

**Implications for Design 20-21:** Based on these findings, a tool for supporting the schema sensemaking step should primarily focus on revealing the ties between the code items (e.g., spreadsheet’s spatial layout and their underlying cell dataflow dependencies), as well as displaying both the code itself (code inspection) and output values (testing) in the context of that structure.

If possible, the tool should also display the information revealed by the others strategems (e.g., specifications) in the context of the main four strategems.

Once again, while these findings come from examining a particular population in a specific environment, they are general enough to be applicable to any debugging environment. For example, Littman found that professional programmers also use similar strategems for systematically comprehending programs: they try to understand the dependencies (dataflow / control flow) of multiple modules (code inspection / spatial) to make sense of them (create a schema) [Littman 1987]. The one main schematizing strategem Littman does not mention is testing. However, based on the popularity of hovering over variables to see their values during control flow (break point) debugging for both professional programmers and IT professionals [Grigoreanu et al. 2009a], we believe output values should also be visible alongside the executed code in order to help users quickly organize the information they have collected.

Another clue to the importance of supporting these strategems for the purpose of schema creation has been the research direction of tools for visualizing hidden
spreadsheet structures. These have focused particularly on visualizations for spreadsheets’ spatial and/or dataflow structures, including tools which highlight spreadsheet areas that seem to create an entity for visualizing broken areas (e.g. [Sajianemi 2000]), tools for visualizing and/or animating dataflow dependencies (e.g., [Igarashi et al. 1998]), and even visualizations which show the dependencies in 3D layers [Shiozawa et al. 1999]. None of these tools support all four (let alone all ten) of the strategems in conjunction and, unfortunately, these studies also do not report empirical data on users’ success in comprehending the spreadsheet or in debugging it. We thus do not know how successful these tools are at helping users create a schema. However, our findings reinforce the need for such tools.

The schema step, while it seemed to be the most problematic for our unsuccessful participants, is of course not the only sensemaking step which needs to be supported through strategy-based tools. For all the rest of the sensemaking steps (e.g., how to successfully find new information, generate correct hypotheses about bug fixes, and evaluate a fix), Table 6 shows the strategems employed successfully by the two successful participants. As before, our threshold for inclusion in the Table was that the strategem needed to be used at least 10% of the time spent in a sensemaking step.

TABLE 6 ABOUT HERE

6. CONCLUSIONS, DISCUSSION, AND FUTURE WORK

This paper is the first to present a comprehensive strategy-based approach for analyzing the usability of end-user programming environments. In particular, we analyzed empirical strategy data at four levels and in two contexts.

Overall, our work revealed the importance of this approach: each level led to a different type of implication for design. Analysis of the data at the lower levels (moves and tactics) led to somewhat more incremental changes to existing features, whereas examining the higher levels (strategems and strategies) led to implications for the design of tools which do not yet exist but should, and also revealed several new opportunities for future research. The two contexts (debugging phase and sensemaking step) helped us understand the purpose for which a strategy item was employed. These two dimensions were also important to examine since a strategem might currently be useful in one context (e.g., feedback following for bug finding) but not another (e.g., feedback following for bug fixing).

This comprehensive understanding of strategy items led to 21 new implications for the design of end-user debugging tools and to six implications for future research. While these implications can be addressed individually, we believe that they should all be taken into account in the design of any one tool, since many of them are complementary.

For example, let us consider how all the implications can be taken into account in the design of a to-do listing tool for end-user debugging environments. Implications for
design 5-6 and 8-11 would be central to the design of a tool to directly support this strategem, since they are all about the missing to-do listing strategem in this study. These implications reveal the importance of a tool which automatically generates a list of consolidated (by some measure of similarity, such as consistent formulas) to-do items. The elementary functionality of this tool would be to keep track of which items are unchecked vs. checked vs. marked for later follow-up, without losing the original cell formatting. In the context of each item, additional information should be provided either as editable data or as subtasks for that particular task item. The tool should not force a particular order in visiting unchecked items, since it should be flexible enough to support both comprehensive and selective debugging strategies.

The remaining implications, while more secondary to the design of this particular tool, provide additional insights into it. The tool could superimpose the status of to-do items onto the spreadsheet, highlighting areas of similar formulas and their status (implication 1). Certain items which have a high likelihood of containing an error, such as an inconsistent formula, could be highlighted in the list (implication 2). An item could be any piece of code executable by the environment (implication 7), and related information about each item (implications 15-18) should be displayed, especially spatial and dataflow dependencies to help users organize the data they collect (implications 19-21). Furthermore, the tool should facilitate understanding of inter-worksheet dataflow relationship ties and help navigating them (implication 3). It should be flexible enough to not render useless any of the successful overall strategies the user might want to employ (implications 12-14). Code inspection for the purpose of viewing related code to help fix bugs could be taken into account by displaying “related formulas” (for example, ranked by physical proximity or syntactic similarity) within the context of each task item (implication 4).

This is an example of how all 21 implications for design can be supported in the creation of a to-do listing tool. However, if the user were planning on building a better code inspection tool or feedback following tool instead, a different facet of these same implications could be considered in its design.

Any empirical study also has limitations. Two such limitations result from having picked a particular environment, population, and task. First, the choice of spreadsheet used and inserted bugs could have led to lower internal validity, since they might have affected the users’ strategies. To lower this risk, we harvested bugs created by real end-user programmers working on this same spreadsheet. Second, our participants might not be representative of the overall end-user programming population, since we only analyzed four participants’ data in detail and since the study was conducted in only one end-user programming environment, Excel. To help address this external validity problem, we have discussed the ties between our results and others’ work from different environments in our results sections. Thanks to that triangulating evidence, we believe that both the analysis methods employed here and the implications for design generalize across debugging environments.

Finally, this comprehensive exploration of strategy usage revealed several opportunities for further research. First, we will build a tool which addresses some of the
implications for design listed here. The tool’s evaluation will allow us to measure the effectiveness of these implications in helping end-user programmers debug spreadsheets. Other implications for further research involve a better understanding of the very popular code inspection strategem in different contexts, of the ties between the prior experience strategem and reuse, and of how to more directly support hypothesis creation, presentation, and reevaluation in end-user programming environments.

This strategy-based approach to usability has revealed many improvements to even one of the oldest and most popular end-user programming environments: Microsoft Excel. For future studies, applying this same approach and a different experiment setup in studying the usability of other end-user programming environments (e.g., a field study with Windows PowerShell) may reveal new implications for the design of strategy-based end-user debugging tools.
NOTES

Acknowledgments. We thank Susan Wiedenbeck for insights and helpful discussions of this work over several years, Peter Pirolli for feedback on our mapping of his Sensemaking model to end-user programming, Kyle Rector for her help with the content analysis for this study, and Jill Cao for her insights. We also especially thank the participants of our study.

Support. This work has been supported in part by the EUSES Consortium under NSF 0325273, by NSF 0917366, by AFOSR FA9550-09-1-0213, and by an IBM Faculty Award.

Authors’ Present Addresses. Valentina Grigoreanu: School of Electrical Engineering and Computer Science, Oregon State University, Corvallis, OR 97331. Email: i.valentina.g@gmail.com, Margaret Burnett: School of Electrical Engineering and Computer Science, Oregon State University, Corvallis, OR 97331. Email: burnett@eecs.oregonstate.edu. George Robertson: PO Box 177, Northeast Harbor, ME 04662. Email: ggr@microsoft.com.

HCI Editorial Record. (supplied by Editor)
REFERENCES


Burnett, M., Cook, C., Pendse, O., Rothermel, G., Summet, J., and Wallace, C. (2003) End-user software engineering with assertions in the spreadsheet paradigm, In


FIGURE AND TABLE CAPTIONS

Figure 1. (Top) A thumbnail of the description handout of the grade-book spreadsheet. (Bottom) Blowup of description Box F.

Table 1. Number of correct and incorrect bug finds and fixes by our most and least successful participants. (The attempts are counted here. For example, the Successful Male made eight incorrect attempts to fix one bug.)

Table 2. This is a sample of promising tactics employed by our participants (SF, SM, UF, and UM), which were accomplished through a series of ineffective moves.

Figure 2. In Excel, when a cell’s dependents are on a different worksheet, a table icon is displayed.

Table 3. Past empirical work on end users’ debugging strategems with ties to gender. A * means statistically significant gender differences have been reported in past work. A + means in-depth qualitative observations of the strategic activity have been reported in past work.

Figure 3. Total minutes the four participants allocated to each strategem.

Table 4. Qualitative empirical findings about the debugging strategies employed by our end-user programmers.

Figure 4. Total minutes spent using each strategem at each step of the sensemaking model by the four participants.

Table 5. Open questions for future research implied by Section 5.2’s findings.

Table 6. Participants SF’s and SM’s most common strategems for each sensemaking step. The amount of minutes spent in each strategy is in parentheses. Bold: Appeared in both participants’ top 10%. Italics: This strategy was also used by the other participant at this step, though for less than 10% of the time. Plaintext: The other participant did not use this strategy at all at this step.
FIGURES

Figure 1. (Top) A thumbnail of the description handout of the grade-book spreadsheet. (Bottom) Blowup of description Box F.

Box F: Average of all students’ grades, maximum grade, and minimum grade.
Table 1. Number of correct and incorrect bug finds and fixes by our most and least successful male and female participants. (The attempts are counted here. For example, the Successful Male made eight incorrect attempts to fix one bug.)

<table>
<thead>
<tr>
<th>Participant</th>
<th>Bug Finds</th>
<th>Bug Fixes</th>
<th>Evaluations of Fixes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Correct</td>
<td>Incorrect</td>
<td>Correct</td>
</tr>
<tr>
<td>SF</td>
<td>9</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>SM</td>
<td>8</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>UF</td>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>UM</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Table 2. This is a sample of promising tactics employed by our participants (SF, SM, UF, and UM), which were accomplished through a series of ineffective moves.

<table>
<thead>
<tr>
<th>Tactic</th>
<th>Moves</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1. Finding Formula Cells:</strong> Examining cells to determine which contained formulas.</td>
<td>[SF] Selected a range of cells and then tabbed through it to make sure that none of the selected cells contained formulas. [SF/SM/UM/UF] Using the arrow keys, and unbounded area, to do the same thing.</td>
</tr>
<tr>
<td><strong>2. Finding Inconsistent Formulas:</strong> Figuring out which formulas do not match the pattern of the other formulas in the same area of the spreadsheet.</td>
<td>[SF/SM/UM/UF] Used the arrow keys to traverse what should have been consistent row or column cells, looking for inconsistencies by examining the changes in the formula bar. [SM/SF/UF] Brought up precedents arrows on two or more cells at a time to check that the formulas are consistent. [SF] Wanted to only verify cells with “Inconsistent Formula” warnings. It took her until the very end of the task, with Error Checking running in the foreground, to discover the Options button. At which point she immediately unchecked everything except for “cells containing formulas that result in an error” and “formulas inconsistent with other formulas in the region.”</td>
</tr>
<tr>
<td><strong>3. Viewing Dependencies:</strong> Viewing a cell’s precedents or dependents to understand what feeds into the cell’s content and how it is used further.</td>
<td>[SF/UF/UM] Clicked on a cell to view its dependents. When the precedents or the dependents were on a different worksheet, the arrows pointed from or to a table icon, which none of the participants understood. In all of these situations, the participants would have navigated to the second worksheet. [SF/UM] Instead, only Participants SF and UM did so... And not until minute 32. Neither found the bug nested there. [SM/UF] Never switched to the second worksheet. [SM] An incorrect bug fix resulted in a circular reference to the first 23 rows of the spreadsheet, only denoted by a blue line below the 23rd row, since the other three borders were invisible. Participant SM stated, “Interesting line there.”</td>
</tr>
<tr>
<td><strong>4. Reusing Working Code:</strong> Following the example of already written code to fix a bug.</td>
<td>[SF/SM] Fixed inconsistent formulas by copying the correct formula over the inconsistent ones. [SM] Fixed the toughest bug by finding other formulas which were used in the spreadsheet, looking up further examples of how to use them, and trying to apply them in that context. [UF] Looked for examples in help that were kind of similar to what she wanted to do, but did not know what to search for.</td>
</tr>
<tr>
<td><strong>5. Figuring Out What’s Next:</strong> Looking to the environment for indications of what to try next.</td>
<td>[SF] Clicked on several buttons of the formula auditing bar to find something new to help her find and fix remaining bugs. She spent almost the entire time looking for inconsistent formulas, ignoring other types of bugs. And, she did not hover over the comments embedded in the spreadsheet until she thought she had run out of things to look at. [UF] Looked for examples in help to continue trying to figure out how to fix a bug she did not know how to fix.</td>
</tr>
</tbody>
</table>
Figure 2. In Excel, when a cell’s dependents are on a different worksheet, a table icon is displayed.
Table 3. Past empirical work on end users’ debugging strategems with ties to gender. A * means statistically significant gender differences have been reported in past work. A + means in-depth qualitative observations of the strategem’s use have been reported in past work.

| Strategems [Grigoreanu et al. 2006; Subrahmaniyan et al. 2008; Grigoreanu et al. 2009a] |
|---------------------------------|-------------------------------------------------------------------------------------------------|
| **Dataflow** | Following data dependencies. *  
Males preferred dataflow. *  
Males successful with dataflow. *  
Dataflow tied to males’ success in finding bugs and evaluating their fix. + |
| **Testing** | Trying out different values to evaluate the resulting values. *  
Males successful with testing. *  
Testing tied to males’ success in finding, fixing bugs, and evaluating their fix. + |
| **Code Inspection** | Examining code to determine its correctness. *  
Females successful with code inspection. *  
Code inspection tied to females’ success in finding and fixing bugs. + |
| **Specification Checking** | Comparing the description of what the program should do with the code. *  
Females successful with specification checking. * |
| **Feedback Following** | Using system-generated feedback to guide debugging efforts. * |
| **To-do Listing** | Indicating explicitly the suspiciousness of code (or lack of suspiciousness) as a way to keep track of which code needs further follow-up. *  
Females preferred to-do listing. * |
| **Fixing Code** | Explicitly described strategy in terms of editing code to fix them. *  
Females unsuccessful with the “fixing code” strategy. * |
| **Spatial** | Following the layout of the program in a spatial order. * |
| **Control Flow** | Following the sequence in which instructions are executed. + |
| **Help** | Accessing Help resources such as the software’s built-in help, internet searches, or another person. + |
| **Proceed as in Prior Experience** | Recognizing a situation (correctly or not) experienced before, and using that prior experience as a blueprint of next steps to take. * |
Figure 3. Total minutes the four participants allocated to each strategem.
Table 4. Qualitative empirical findings about the debugging strategies employed by our end-user programmers.

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Females and Males</td>
<td>Suggestive evidence that females’ and males’ success with a strategy differed with different debugging stages (finding a bug, fixing a bug, or evaluating a fix) [Grigoreanu et al. 2009a].</td>
</tr>
<tr>
<td>SF</td>
<td>Comprehensive strategy: traversing a spreadsheet systematically in order to look for bugs, only fixing the ones which do not derail her from her processing [Grigoreanu et al. 2009b].</td>
</tr>
<tr>
<td>SM</td>
<td>Selective strategy: systematically following up on the latest bit of relevant information collected, once a bug is found he stuck with it either until he fixed it or until a new bug was found [Grigoreanu et al. 2009b].</td>
</tr>
<tr>
<td>SF and SM</td>
<td>Consistency checking strategy: Using all eight strategems (especially specification checking, spatial, code inspection, and dataflow) together to make sure that bits of code which should be consistent are indeed so. This strategy was used for both finding and fixing bugs.</td>
</tr>
<tr>
<td>SF</td>
<td>Simple mapping strategy: Combining all eight strategems (especially feedback following, spatial, help, and code inspection) to work backward from error checking messages in finding bugs.</td>
</tr>
<tr>
<td>SM</td>
<td>Causal reasoning strategy: Working backward from the code’s output which seems faulty to find a bug (especially using testing and code inspection).</td>
</tr>
<tr>
<td>SF and SM</td>
<td>Compatibility checking strategy: Strategy employed by our participants to evaluate their bug fixes (especially relying on testing, code inspection, and/or spatial).</td>
</tr>
</tbody>
</table>
Figure 4. Total minutes spent using each strategem at each step of the sensemaking model by the four participants.
Table 5. Open questions for future research implied by Section 5.2’s findings.

<table>
<thead>
<tr>
<th>Implications for Future Research 1-3:</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Why is code inspection such a dominant strategem?</td>
</tr>
<tr>
<td>- How does usage of code inspection differ by sensemaking step?</td>
</tr>
<tr>
<td>- Should designers create one code inspection tool to support all sensemaking steps, or specialized tools for the different steps?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Implications for Future Research 4-5:</th>
</tr>
</thead>
<tbody>
<tr>
<td>- How can prior experience be supported during the early steps of sensemaking: in skimming information and in deciding which information to follow up on?</td>
</tr>
<tr>
<td>- What are the ties between prior experience and the reuse tactic mentioned in Section 3.2.?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Implications for Future Research 6:</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Would it be useful to support all eight strategems as a part of the later sensemaking steps of hypothesis, presentation, and reevaluation?</td>
</tr>
</tbody>
</table>
Table 6. Participants SF’s and SM’s most common strategems for each sensemaking step. The amount of minutes spent in each strategy is in parentheses. **Bold:** Appeared in both participants’ top 10%. **Italics:** This strategy was also used by the other participant at this step, though for less than 10% of the time. **Plaintext:** The other participant did not use this strategy at all at this step.

<table>
<thead>
<tr>
<th>Sensemaking Step</th>
<th>SF: Code Inspection (5.9), Specification Checking (3.6), Dataflow (2.0), Feedback F. (1.6)</th>
<th>SM: Dataflow (3.4), Help (2.4), Code Inspection (2.2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shoebox</td>
<td><strong>Code Inspection</strong> (8.6), Dataflow (2.1), Spatial (1.5)</td>
<td><strong>Code Inspection</strong> (6.0), Dataflow (3.4), Help (1.2)</td>
</tr>
<tr>
<td>Evidence File</td>
<td>SF: <strong>Code Inspection</strong> (2.7), Dataflow (2.1), Spatial (1.5)</td>
<td>SM: Code Inspection (6.0), Dataflow (1.2), Spatial (1.0), Help (1.0)</td>
</tr>
<tr>
<td>Schema</td>
<td>SF: <strong>Code Inspection</strong> (2.9), Spatial (3.1)</td>
<td>SM: <strong>Code Inspection</strong> (2.8), Testing (1.5), Spatial (1.3), Dataflow (1.1)</td>
</tr>
<tr>
<td>Hypothesis</td>
<td>SF: <strong>Code Inspection</strong> (0.6), Spatial (0.1), Dataflow (0.1)</td>
<td>SM: Code Inspection (4.4)</td>
</tr>
<tr>
<td>Presentation</td>
<td>SF: <strong>Code Inspection</strong> (0.3), Spatial (0.3), Dataflow (0.1)</td>
<td>SM: Code Inspection (4.5)</td>
</tr>
<tr>
<td>Reevaluate</td>
<td>SF: Testing (0.4), Spatial (0.2), Code Inspection (0.1)</td>
<td>SM: Code Inspection (0.9), Testing (0.7), Spatial (0.2)</td>
</tr>
<tr>
<td>Environment</td>
<td>SF: Feedback Following (3.5), Help (3.2), Code Inspection (1.8)</td>
<td>SM: Help (3.0), Code Inspection (1.8)</td>
</tr>
<tr>
<td>Common Sense</td>
<td>SF: Prior Experience (0.8), Code Inspection (0.7), Dataflow (0.2)</td>
<td>SM: Help (0.5), Specification Checking (0.3), Code Inspection (0.3)</td>
</tr>
</tbody>
</table>